A Functional Model of Jazz Improvisation

Preprint · June 2019

2 authors, including:

Donya Quick
Stevens Institute of Technology
20 PUBLICATIONS **82** CITATIONS

SEE PROFILE

A Functional Model of Jazz Improvisation

Donya Quick
Stevens Institute of Technology
donyaquick@gmail.com

Kelland Thomas
Stevens Institute of Technology
kthomas3@stevens.edu

Abstract

We present a model of jazz improvisation where short-term decision making by each performer is modeled as a function from contexts to music. Contexts can be shared, such as an agreed-upon chord progression, or they can also be private—a current state for each musician. We formalize this model in Haskell to generate potentially infinitely long jazz improvisations, and we have also used the same model in Python to support real-time human-computer interaction through jazz.

Keywords generative music, jazz, improvisation, functional programming

1 Introduction

Modeling jazz improvisation in a sufficiently general, yet computable way is an open problem. In this paper, we present a functionally-inspired view of how traditional jazz improvisation and the flow of information between musicians can be modeled. Musicians are modeled as a combination of two things: (1) a function from contexts to music and (2) a private context or state. We have implemented our high-level model in two languages: Haskell and Python. We have primarily explored bossa nova and styles with a walking bassline in our two implementations.

The Haskell implementation, which is detailed here, permits the computer to improvise with itself and produce potentially infinitely long pieces of music. It serves as a formalism of the problem, and we are actively augmenting it to support new styles of music.

Our Python implementation was directed towards human-computer interaction and treats the human as simply another function, just as the other computer musicians are modeled. While the Python implementation is in a less well-structured state than our Haskell implementation, and therefore covered in less detail, we are currently using the Python version as a way to test human reactions to different generative algorithms—an important metric for evaluating algorithmically generated music.

2 Related Work

Improvisation in a jazz ensemble setting is conditioned on certain contexts. In orthodox modern jazz[1], harmonic and rhythmic constraints are decided upon and shared by all performers in advance[10]. These constraints constitute a shared context for that performance. Aspects of the musical performance that are shared by all musicians include the harmonic progression, meter, and tempo. In addition to shared contexts, we can also assume that each musician has some internal model that helps to govern what he/she plays next.

There are several studies of the cognitive processes employed in musical improvisation[10, 24]. This work highlights one of the key difficulties in improvisational music: unlike other forms of composition, where the work can be revisited and altered in multiple passes, an improvising jazz musician must make decisions quickly from a finite amount of working memory and cannot go back in time to make corrections. This is much like what is called a *transducer* in automata theory: a recursive function that produces output while reading input.

An improvising musician's internal generative model should produce musical statements that meet the constraints provided by the shared musical context. Debate in the literature exists about how this is most effectively modeled. This internal generative model may be influenced by biological priors, learned sets of rules, or a combination of the two [10]. The use of repeated melodic patterns in improvised output is well documented[2], and one hypothesis posits that performers employ patterns as pre-learned structures [17]. An alternative hypothesis involves the use of rules, like algorithms, to generate novel musical statements without resorting to pre-learned patterns sitting in memory [10]. Norgaard suggests that these are both happening in advanced improvisers [13], but subsequent computational experiments suggest that the former hypothesis is more likely [14]. Regardless of the source of the patterns, this work collectively suggests that improvisational behavior can be modeled as the episodic production of collections of notes. We take a similar view of music generation in the models presented in this paper.

[1]Excluding practices and traditions often termed "free jazz", where no or very few constraints are decided on or shared prior to beginning a performance.

ACM SIGPLAN Workshop on Functional Art, Music, Modeling, and Design, 2019

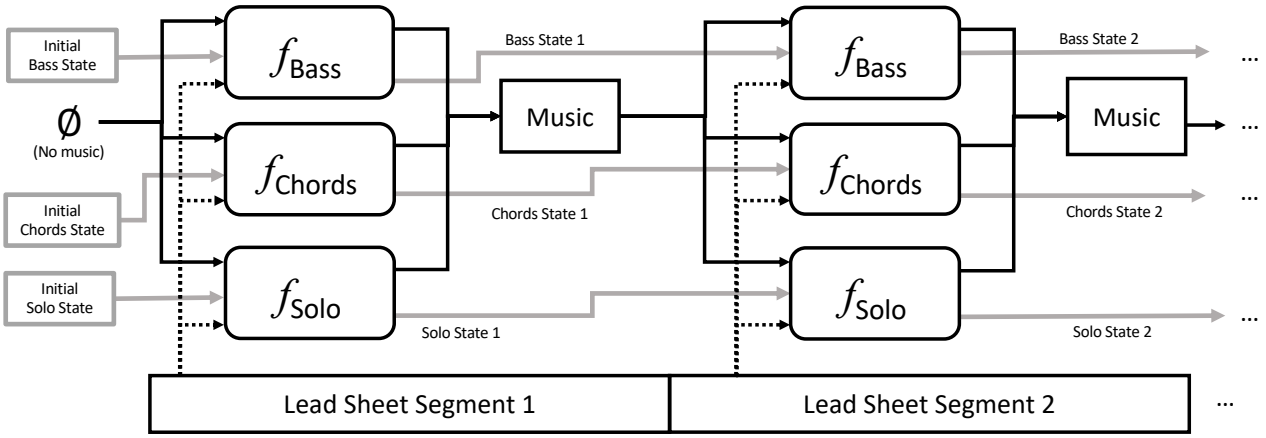

Figure 1. Illustration of our model for improvisation over lead sheet segments. Gray lines indicate passing of state information, black lines indicate observation of produced music, and dashed lines are observation of the lead sheet segment. We assume that all parts "hear" the music produced by all other parts, while each part's generative function only receives state information relevant to itself.

In contrast to the idea of generating collections of notes in a single step, many statistical models for music have focused on note-to-note transitions. Markov chains have historically been a popular model for music analysis and generation [4, 16, 18, 26]. The states involved in these models are typically single notes or chords rather than sequential collections of notes. While a useful analytical tool and often successful for short-term generation, long-term structure is more difficult to model, and Markov-based generative models typically only make judgments based on history rather than planning towards a specific goal. The notion of long-term structure is somewhat different in an improvisational setting than for offline composition, but it still manifests in tasks such as following the constraints of a lead sheet, remembering motifs used by other musicians, and deciding when and how to insert them at a later time. While long-term structural issues can be mitigated with longer history lengths in models such as variable-length Markov chains [23], the cognitive models described previously suggest that the notion of states these models assume may not be well-suited to how musicians create improvisational music.

Neural nets are also increasingly popular in generative music[1, 6, 15], and have the benefit of being a rapid generative model once trained. However, neural nuets often suffer from similar problems to Markov chains for achieving long-term coherency, although some attempts to model long-term structure do exist, including for jazz [5]. Once trained, the adaptability of neural nets to other situations is also problematic - there is typically no way to simply insert a new rule or definition without providing additional training data. Our framework aims to address some of these long-term issues while also avoiding over-fitting the model to a particular style.

A number of grammatical approaches have been employed in music analysis and generation [7, 12, 20–22]. Grammars can solve some of the long-term structural problems exhibited by Markov chains and neural nets, making them appealing for modeling offline music composition. However, they are not always suitable for real-time usage. While a fast generative algorithm may exist for a given category of grammars, a similarly rapid parser may not. For probabilistic grammars, which are a popular generative strategy, learning production probabilities is often a computationally non-trivial task. These issues potentially problematic if the same model is to be used both in analytical and generative capacities to, for example, analyze a solo and respond to it appropriately.

There has also been substantial work in the area of generative algorithms for jazz, with a focus on creating tools for musicians [9, 11]. While useful for generating novel musical content (an idea generator for composers) and as practice tools in a variety of styles (for the isolated musician who needs a band), these systems do not support dynamically changing contexts in a way that would allow the computer musicians to adapt on-the-fly to each other and human participants over the span of an individual piece. While there has been some work in the area of trading solos between a human and computer[3], it is not concerned with generatively modeling other parts of the music like bass and harmony. Our model aims to address shifting contexts for all of the parts, such that generative algorithms for bass, harmony, and solos, can all interact under the same framework. While not directly addressed in this paper, drum parts can also be supported.

```
type PCNum = Int
type Scale = [PCNum]
data ChordCtxt =
  ChordCtxt{sym::String, scale::Scale}

cM7 = ChordCtxt "CM7" [0,2,4,5,7,9,11]
dm7 = ChordCtxt "Dm7" [2,4,5,7,9,10,0]
g7 = ChordCtxt "G7" [7,9,11,0,2,4,5]
```

Figure 2. Types for our implementation of chord contexts and examples of definitions for three chords. The first chord, CM7, corresponds to a C-major scale. Dm7 represents a chord from the D minor scale, and G7 is a dominant 7th chord corresponding to the G Mixolydian mode (a major scale with a lowered seventh).

```
data Segment a = Segment{
  chordCtxt :: ChordCtxt,
  segOnset  :: Onset,
  segDur    :: Beat,
  ... }

seg1  = Segment cM7 (1,0) 4 ...
seg2  = Segment dmM7 (2,0) 4 ...
seg3  = Segment g7 (3,0) 4 ...
seg4  = Segment cM7 (4,0) 4 ...
leadSheet =  [seg1, seg2, seg3, seg4]
```

Figure 3. The Segment type and an example definition of a 4-segment lead sheet. It represents a I-ii-V-I progression in C-major. Some fields of the Segment definition are omitted since htey are not used by the examples in this paper.

3 Model

We propose a model of improvisational jazz interactions where each musician's decision-making process is modeled as a function from musical context to performed music over short segments of music (typically 1 measure or less). We define several kinds of musical context: the lead sheet, a performer's internal state, and observed music. Figure 1 shows the flow of information in our model.

The *lead sheet* defines a shared set of harmonic assumptions across musicians. It is a shared context amongst all participants. This need not be limited to formally notated lead sheets - it can simply be a repeating chord progression that all parties have agreed upon before playing, such as ii-V-I in G-major. The lead sheet informs both what the current harmonic context is, as well as what changes in that context are coming in the future. A lead sheet is divided into *segments*, such one segment per measure. Segments may also start/end at other important musical features, such as key/chord changes or boundaries of fixed compositional elements.

Each performer also has an *internal state* representing decisions made that are not known to the other performers. This state, essentially a private context for decision-making, will include a mixture of short term memory and future planning. For example, knowing that a key change is coming soon, a bass player may need to plan a series of notes that is largely within the current scale, avoids large jumps, and lands on the root of the new key on the beat at which the change occurs. Similarly, when planning which pitches to play for the next chord, a pianist or guitarist will need to consider only options that are reachable for his/her particular physical limitations (like hand size). Soloists may have a plan for a motif to introduce in a particular upcoming section.

Finally, each musician also hears *everyone's music*, or the history of what has been performed. This common set of observations is exclusively in the past (by the time musician A hears musician B start playing a note, that event has already happened). While a more continuous model is perhaps most cognitively accurate, it also presents problems from the standpoint of creating real-time applications, where too-frequent calls to generative and analytical algorithms can create an excessive computational burden that negatively impacts performance. Therefore, we take a simplified model where the history is aggregated over the relatively short segments that comprise our lead sheets.

Figure 1 only features one of each kind of part (bass, chords, solo). This is the strategy used in our Haskell implementation, which is described in Section 4. However, the same model is extensible to other scenarios and also to human-computer interaction. In a scenario such as *trading fours*, a task where two musicians alternate short solos, we could have f_{solo1} and f_{solo2} that become active over alternating collections of lead sheet segments. Our Python implementation (see Section 6), supports substitution of human input for one of these functions, retaining the same general flow of information.

Our model is only intended to capture improvisational music, not static compositions such as fugues and sonatas—styles that have constraints well beyond that of improvisational music. Trying to apply a model intended for improvisation to a non-improvisational style is likely to result in only a surface-level similarity to the intended target, with deeper features of the intended style being lost. While our model can be generalized to other kinds of improvisational music and is not strictly limited to jazz, it is best suited to music that is strongly based around real-time decision-making.

4 Haskell Implementation

We provide an implementation of our high-level model in Haskell. We first define a system of data and function types to formalize improvisational behavior. Assuming the existence

of appropriate function definitions for the short-term behavior of individual parts, we then supply a generative algorithm that can "run" the system. Examples of part functions are included in Section 5.

Our Haskell implementation makes use of some music representation features from the Euterpea library[8], which provides representations for musical structures along with export and playback options using MIDI. We make use of the following types and data structures from Euterpea, which define a tree structure for music:

```
type AbsPitch = Int
type Dur = Rational
data Primitive a  =
  Note Dur a |  Rest Dur
data Music a  =  Prim (Primitive a)
    |  Music a :+: Music a
    |  Music a :=: Music a
    |  ...
```

Euterpea's *(:+:)* constructor, sequential composition, means to play two musical subtrees sequentially. Similarly, the *(:=:)* constructor for parallel composition means to play two musical subtrees in parallel, such that they have the same start time (end times may be different). Leaf nodes in trees are either notes (*Note*) or rests (*Rest*). The type variable, *a*, in the *Music* data structure is used for storing pitch information, and sometimes other values like volume. In this paper, we will only use a pitch number (an integer) representation for pitch and will ignore other possible parameters. Therefore, a *Note* leaf node will contain a duration and a pitch number. The *AbsPitch* type means "absolute pitch" and refers to pitch numbers. We use the MIDI standard of pitch numbers from 0 to 127.

In Euterpea, which has no explicit representation for time signatures, durations are represented in terms of measures of 4/4, such that one measure in that time signature has the value 1. In contrast, the generative framework presented in this paper represents lead sheet segment duration in terms of beats, and it also allows specification of a time signature for each segment. The code in the following sections involves conversions between these two methods of measuring time: one beat of a segment in 4/4 has the value 1, while the equivalent Euterpean duration value is 1/4.

4.1 Lead sheets

A lead sheet can be viewed as a partial specification of a performance, and some sections of the music will be more concretely defined than others. The common feature across all parts is the notion of a *chord context*. Chord contexts in sheet music are typically indicated using chord symbols such as Cmin7 or Cm7 (both read as: "C minor seventh chord"). Each one of these symbols implies a particular scale.

In our implementation, a scale is a list of pitch classes, beginning with the root. We represent pitch classes as the integers 0-11 for C through B. For example, the C-major scale would be represented ast [0,2,4,5,7,9,11]. Enharmonically equivalent pitch classes share the same pitch class number, or *PCNum*. A chord context, then, is a chord symbol (String) and the scale it implies. Figure 2 shows these type definitions and examples of definitions for specific chords.

```
type History a = [(PartType, Music a)]
type PartFun a s =
  s -> -- part state
  Segment a -> -- current segment
  Maybe (Segment a) -> -- next segment
  History a -> StdGen -> (StdGen, s, Music a)

data JazzPart a s = JazzPart {
  partType :: PartType,
  instr :: InstrumentName,
  partFun :: PartFun a s,
  state :: s}

type JazzBand a s = [JazzPart a s]
```

Figure 4. The type of a part function and jazz band in our framework.

Each chord context lasts for some duration of time. Chord contexts often span whole measures, but not always. Sometimes, they may start in the middle of a measure or even last for only a single beat. We will refer to spans of a lead sheet falling under a single chord context as *segments*. Each segment has an onset at a particular measure and beat, and has a duration in beats. A lead sheet, then, is simply a list of segments. Segments may also contain other information, such as the current time signature and whether there are fixed musical elements to be played verbatim. However, these extra kinds of information are not featured in the examples in following sections and, therefore, have been omitted from the definitions given in this paper. Figure 3 shows an example definition of a lead sheet.

It is worth noting that lead sheets in our system do not have to be finite—nor must they be repeating. In fact, the generative algorithms we have tested with our system to date all work on randomized lead sheets. While this is perhaps not entirely applicable to most human-made music, it illustrates the adaptability of our approaches.

4.2 Parts

Each instrumentalist can be considered a *part* in our model. As previously described, the behavior of a part is modeled as a function from a collection of contexts to musical output. Several contexts are shared: the current lead sheet segment, which lead sheet segment is coming next, and what everyone in the group has played up to the current point in time.

```
runSegment :: JazzBand a s -> History a ->
  Segment a -> Maybe (Segment a) ->
  StdGen -> [(StdGen, s, Music a)]
runSegment [] h seg1 seg2 g = []
runSegment (jp:jps) h seg1 seg2 g =
  let (g', st, m) =
        partFun jp (state jp) seg1 seg2 h g
      m' = instrument (instr jp) m
  in  (g', st, m') :
      runSegment jps h seg1 seg2 g'
```

Figure 5. Function for generating a band's music over a single lead sheet segment. The *instrument* function, which is imported from Euterpea, sets the MIDI instrument of a Music value (which is piano if unspecified).

Figure 4 shows the type signature expected for a part function, or *PartFun*. Since we want to allow the possibility of stochastic decision making, we also thread a random generator of type *StdGen* through our generative functions[2]. A *PartFun*, therefore, takes the current state for the part, the current lead sheet segment, the next lead sheet segment (if one exists), the musical history, and a generator. The function will return a new generator, an updated part state, and the music to be played during the current segment. State information is simply represented as an open-ended type variable, *s* (discussed more in section 4.3). Section 5 contains example function implementations for different jazz features using the *PartFun* function type.

4.3 States

States are left as unconstrained type variables in this system. This is because the kind of information that needs to be stored in the state for different parts is an area of ongoing research. For some styles, state information can be relatively simple, while others require more complex states. Multiple data types have been used in experimentation, but the same generative framework can be applied regardless of the type of state used. Typically, a state should be a data type that has constructors to store information relevant to the various parts. Some example state definitions are given in Section 5.

In our implementation, each part is associated with its own state information. However, for cohesiveness within the generative system, the data type used for state information must be the same for all parts. This means that a given part will not necessarily utilize all possible constructors for the state type it uses. For example, a walking bass may require a state with a constructor that stores the target pitch for the start of the next lead sheet segment. However, a pianist playing chords might have no use for that kind of information. Determining which constructors make up a sufficiently generalized yet also sufficiently detailed state is an area of ongoing work. It is also an issue that does not not strongly affect implementation in other languages. For example, an implementation in Python can simply treat the state as a list of heterogeneously-typed objects.

[2]While this passing of random generators can be "hidden" in monadic contexts to simplify some of the type signatures, we have chosen to make the use of random generators explicit and avoid encapsulating the entire implementation within other monads. This is purely a design choice, and we may yet monad-ify the implementation in the future once our collection of generative functions for parts is in a more advanced state.

```
runBand :: JazzBand a s -> History a ->
  LeadSheet a -> StdGen -> Music a
runBand [] h segs g = rest 0
runBand jb h [] g = rest 0
runBand jb h (seg1:segs) g =
  let seg2 = if null segs then Nothing
             else Just (head segs)
      result =
        runSegment jb h seg1 seg2 g
      (gs, states, ms) = unzip3 result
      jb' = zipWith (\st jp -> jp{state=st})
            states jb
      h' = zip (map partType jb) ms
  in  foldr1 (:=:) ms :+:
      runBand jb' h' segs (last gs)
```

Figure 6. Function for "running" a band over an entire lead sheet.

4.4 Generative Function

The generative function iterates over segments of a lead sheet. For each segment, we need to run the generative function for each performer. The function performing this task is called *runSegment*, shown in Figure 5. We then define a function called *runBand*, shown in Figure 6, which calls *runSegment* over each segment of the lead sheet. Note that this implementation is tail-recursive, so it can be used on infinitely long lead sheets and produce infinitely long music.

To use this function, we must first define a jazz band with some parts. Each part has its own initial state.

```
myBand = [
  JazzPart Bass AcousticBass bassFun state0,
  ...]
```

We can then "run" the band on the lead sheet defined in Figure 3 with a random number seed of our choosing. Below, a seed value of 5 is used.

```
m :: Music AbsPitch
m = runBand myJB [] leadSheet (mkStdGen 5)
```

5 Examples

Here we present several examples of part implementations for our framework and general descriptions of algorithms we

have implemented using the described models. For the sake of concision, only the simpler cases are shown with their corresponding Haskell implementations. More advanced algorithms, such as those used in our interactive Python implementation are described in more general terms. Our complete Haskell code and more information on our Python framework is available online[3].

Not all of the features supported by the high-level generative framework are utilized in our examples of part implementations. The high-level formalization developed in Section 3 and its corresponding Haskell implementation in Section 4 were designed to be very general. Their usage with new styles and more complex generative algorithms that more completely utilize its features is an area of ongoing work. We also show examples of musical output from two algorithms that were constructed according to the model in Figure 1, but which used different data structures to those presented in Section 4. Rewriting of those algorithms to use Section 4's data types is an ongoing effort.

For examples including code, we will make use of some Euterpea types and functions:

```
note :: Dur -> a -> Music a
note d  = Prim (Note d p)
rest :: Dur -> Music a
rest d = Prim (Rest d)
```

The *note* and *rest* functions are shorthands to create Notes. We also use several duration shorthands, *qn*, *en*, and *dqn*, which indicate the values quarter note, eighth note, and dotted quarter note respectively. Recall that we will be only using pitch numbers, or *AbsPitch* values, for the "a" type variable in the definitions above, and therefore will be working exclusively with music values of type *Music AbsPitch*.

5.1 A Simple Bossa Nova Bass

Figure 7 is a deterministic bossa nova bass function that is one of the simplest use cases for this framework. For as much duration as the current segment contains, it produces the classic bossa nova pattern over the current scale's root and fifth. The state in this case is not used, and therefore can simply be a type variable. This makes the function applicable to any state type.

Output from this stateless bass function in combination with a similarly stateless strategy for chords is shown in Figure 8. However, both the state-less bass and state-less chords will sound rather repetitive compared to algorithms that maintain a state and take history into account in order to make more complex decisions. Nevertheless, it serves as a kind of bass case to illustrate use of the larger generative framework.

[3]More information, source code, and example recordings are available on our website: www.donyaquick.com/generative-jazz

```
bassFun :: PartFun AbsPitch s
bassFun s seg1 seg2 hist g =
  let p1 = 36 + (head $ scale $ chordCtxt seg1)
      p2 = p1 + 7
      mPat = note dqn p1 :+: note en p2 :+:
             note dqn p2 :+: note en p1
      m = trimTo seg1 (forever mPat)
  in  (g, s, note (segDur seg1 / 4) p1)

trimTo :: Segment a -> Music a -> Music a
trimTo seg m =
  cut (segDur seg) $
  remove (snd $ segOnset seg) m
```

Figure 7. A state-less bossa nova bass implementation, where *forever*, *cut*, and *remove* are functions from Euterpea that respectively repeat a music value forever, preserve only a specified amount from the beginning, and delete a specified amount from the beginning.

5.2 A Walking Bass: Looking Forward

A walking bass line provides an example of the need to plan ahead for the next lead sheet segment. It is common for the bass's pitch on the first beat of a key change to be the root of the new scale. In order to effectively "walk" to this new location, we must decide where it is before plotting intermediate points.

To implement this within our Haskell framework, we use a state that is either null or stores the upcoming root pitch that we must reach by the start of the next segment. Figures 9 and 10 show the important aspects of this implementation. The approach involves finding the pitch spaces corresponding to each segment's scale, which is to say the set of pitches in each segment that conform to its harmonic context. The pitch space of the *next* segment (*seg2*) is used to choose the next root pitch, *nextR*. Then, the state's root pitch is used as a starting point from which to plot a path towards *nextR* by walking through the pitch space corresponding to the current segment (*seg1*). Filling each beat in the current segment left to right, the algorithm will choose an intermediate pitch between the previous pitch and *nextR* if there is room to do so, otherwise it will choose a pitch that is near *nextR* (either above or below by some amount). Finally, *nextR* is passed to the next iteration to become the starting pitch of the next segment.

Figure 11 shows an example of the music produced from this method. More complex versions of this approach can also introduce chromaticism and ornaments into the path-finding process. Figure 12 shows an example of output from a more complex walking bass algorithm that uses the same general strategy.

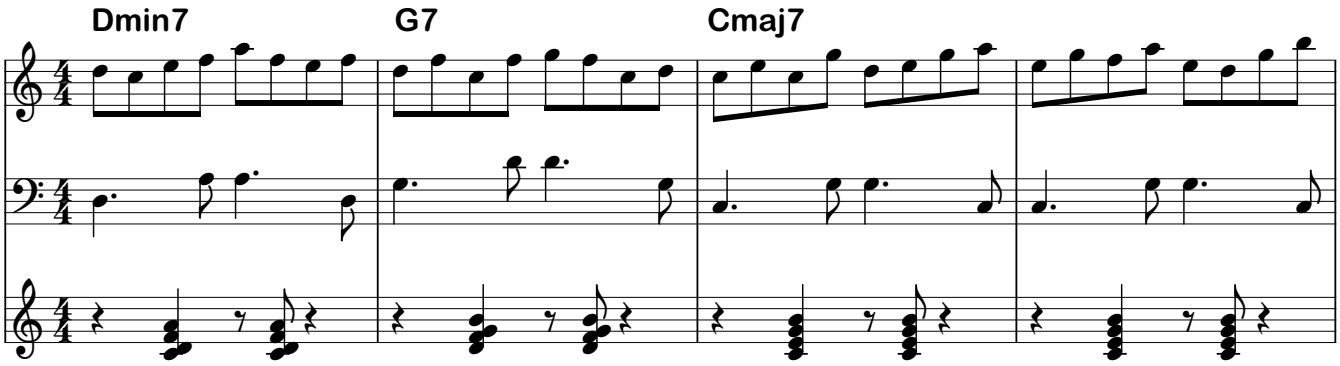

Figure 8. Output from a bossa nova algorithm where the harmony and bass generators do not utilize a state. The solo algorithm utilizes a state for the purpose of keeping track of its last pitch, thereby ensuring smoothness between segments of the lead sheet.

5.3 Generating Solos: Looking Backwards

Soloing algorithms demonstrate the need for looking at the performance's history, or at least a recent subset of it, particularly in scenarios where there is more than one soloist. When two human soloists alternate improvised melodies, they typically incorporate material from previous iterations in their response in addition to adding new material. Musically, this is a way of saying "I heard you, and here are my ideas." We have explored soloing algorithms in our interactive Python implementation with the following behavior:

For each lead sheet segment:

1. If it is not the soloist's turn, remain silent. If it has just become the soloist turn, scan the history for input from other soloists and extract motifs from it.
2. Generate a response as a sequence of chunks, where a chunk is a short collection of notes. With some probability, use a chunk of notes from the observed collection of motifs, otherwise generate a novel chunk.

Currently, we use a random walk within the current scale to generate novel sections. Examples of this is shown in Figures 8 and 13. Figure 14 shows a very simplistic example of chunk-based responses where the responses contain no novel material. Most of the soloing algorithms we have implemented involve direct reuse of portions of the user's pitch sequences. We are currently developing approaches that identify more musically meaningful motifs and that utilize more abstract notions of motifs, such as identifying patterns of pitch intervals that can be reused in transposed settings along the lines of the models proposed by Norgaard [14].

5.4 Generating Chords

Harmony generation at the level of an expert musician is a non-trivial problem while maintaining novelty/variation over time and adhering to real-time constraints. As a result, the harmony implementations we have explored to date in both our Haskell and Python implementations are not terribly complex—we sacrifice some novelty and musical finesse for simplicity and speed. We have primarily used two simple algorithms: generating chords within a small range that are playable with one hand on a piano where each chord is never very far away from the others, and generating chords within a larger range by random pitch selection (not necessarily being playable with one hand, but typically playable with two hands when the pitch range is constrained to two octaves or less). While these produce interesting results, we recognize that they do not capture certain critical elements of decision making in jazz, such as planning the voice-leading between chords and considering how different voicings of the same chord may affect its perception within the larger harmonic context. We are actively investigating algorithms to handle these additional features.

Kulitta [19], a library for automated music composition in Haskell, presents an implementation of chord spaces[25] that can be adapted to jazz. These spaces are time-consuming to generate but significantly faster to traverse, meaning that they can be used in real-time as long as they are generated and loaded into memory before real-time constraints are imposed. We have also found that generating partial chord spaces on the fly to find nearby chords is a viable strategy. This is applicable to instruments like piano where there is a limit to how rapidly a pianist can move his/her hand across the keyboard, typically opting for chords that are close to what was played previously.

6 Interactive Python Implementation

In addition to implementing our model of jazz improvisation in Haskell, we have also implemented the same ideas in Python. Our Python implementation supports real-time interaction with a human musician, where the machine will respond to the human with a mixture of borrowed and novel

```
data WalkingState =
  NextRoot AbsPitch | NullState
bassRange = [36..50] :: [AbsPitch]
toSpace :: [AbsPitch] -> [AbsPitch]
toSpace scale allPs =
filter (\p -> elem (mod p 12) scale) allPs

wBassFun :: PartFun AbsPitch WalkingState
wBassFun NullState seg1 seg2 hist g = ...
wBassFun (NextRoot r) seg1 Nothing hist g = ...
wBassFun (NextRoot r) seg1 (Just seg2) hist g =
  let scale1 = scale $ chordCtxt seg1
      scale2 = scale $ chordCtxt seg2
      pSpace1 = toSpace scale1 bassRange
      pSpace2 =toSpace scale2 bassRange
      roots2 =
        filter (\p -> mod p 12 == scale2 !! 0)
        pSpace2
      (g1, nextR) = choose g roots2
      beats = round (4*segDur seg1)
       (g2, pitches) =
         walk beats pSpace1 r nextR g1
      bassLine = line $ map (note qn) pitches
  in  (g2, NextRoot nextR,
      cut (segDur seg1/4) bassLine) where
```

Figure 9. Haskell implementation of a simple walking bass algorithm, where *choose* is a utility function that stochastically selects an item from a list and the *walk* function is shown in Figure 10. The first two lines of the *wBassFun* function are omitted since their behavior is comparatively uninteresting. In the first case, because there is no state information, the function will simply create a state for itself and then recurse. In the second case, because the second segment is *Nothing*, the end of the piece has been reached and only a single note is played.

```
walk :: Int -> [AbsPitch] -> AbsPitch ->
    AbsPitch -> StdGen -> (StdGen, [AbsPitch])
walk 0 pSpace p1 p2 g = (g, [])
walk i pSpace p1 p2 g =
  let (g2, pMid) = makeStep pSpace p1 p2 g
      (g3, ps) = walk (i-1) pSpace pMid p2 g2
  in  (g3, p1 : ps)

makeStep :: [AbsPitch] -> AbsPitch ->
    AbsPitch -> StdGen -> (StdGen, AbsPitch)
makeStep pitchSpace p1 p2 g =
  let pH = max p1 p2
      pL = min p1 p2
      midPs = filter (\p -> p<pH && p>pL)
           pitchSpace
      nearPs = filter (\p -> p<pL+7 && p>pL-7
           && p/=pL && p/=pH) pitchSpace
      ps = if null midPs then nearPs else midPs
  in  choose g ps
```

Figure 10. One possible implementation of the *walk* function called by the code in Figure 9. The goal of *makeStep* is to find an intermediate pitch meeting certain criteria: if the endpoint pitches, *p1* and *p2*, are far enough apart, then a member of the pitch space that is between them is selected. Otherwise, a nearby, but different pitch is selected. In this case, we define "nearby" as being within 7 half steps or semitones (an interval of a perfect fifth).

material. To date we have primarily explored basic recombinatory strategies for echoing a user's solos, but more sophisticated methods of pattern-extraction are under development for use with pattern-based generation algorithms (the subjects of specific motif detection and pattern-based generative algorithms are both beyond the scope of this paper). We have used our Python implementation in live performances and are currently using as as a way to evaluate the human perception of different generative algorithms for solos.

Time-constraints on generation are non-trivial when performing music in a real-time interactive scenario with a human participant, particularly at faster tempos. Under these conditions, it is not necessarily feasible to transition immediately from observing all of the humans input over one segment to generating a coherent response immediately in the next segment. If decision-making is left too late, we risk an audible pause in the music while the computer stops to "think" before it can begin playing again.

Parallelism and asynchrony are sometimes required to allow the computer enough time to make decisions without creating audible pauses or time leaks. In our real-time Python implementation, we allow the computer to start running its functions for each part over the next lead sheet segment a small amount of time before the current segment has actually finished, ensuring that the results will be ready in time. The general flow of information remains the same as in Figure 1, but there is a small overlap on the order of 10-250 milliseconds between the music playing and when the functions are executed. This means that the end of the current segment's music is still playing while the computer quickly creates music for the next segment, thereby ensuring a seamless result. This overlap of the generative phase with the end of the current segment's music typically only results in only one beat's worth of the user's input not being considered during generation. Of course, the exact degree of overlap needed (and therefore the amount of user input that becomes inaccessible during generation) will depend on the tempo

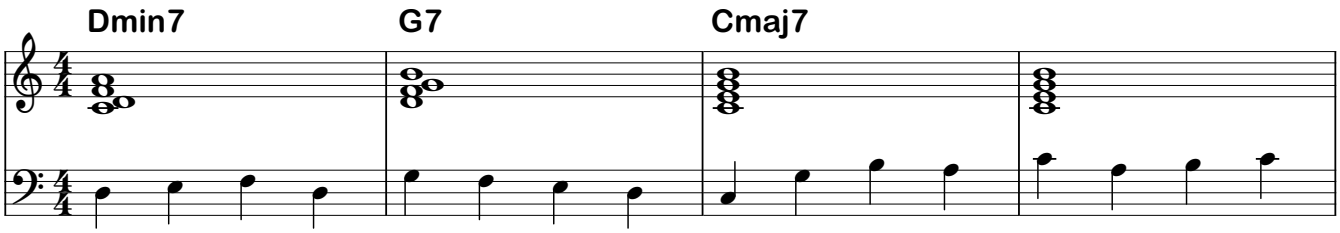

Figure 11. Output from a simple walking bass algorithm with chords added.

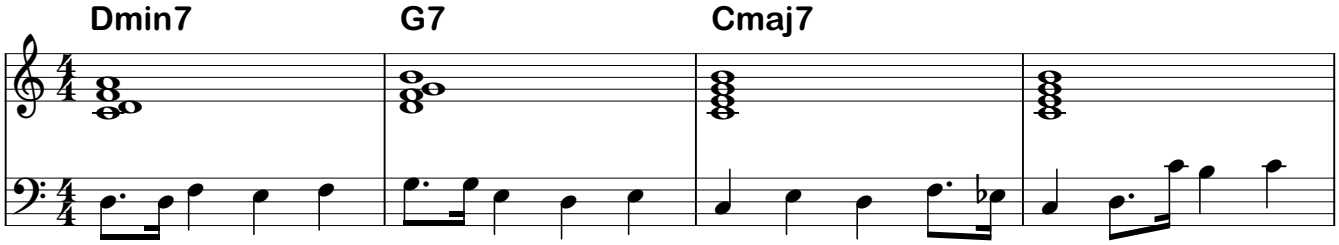

Figure 12. Output from a more complex walking bass algorithm that features chromatic steps and ornaments.

and complexity of the generative algorithms used for each part.

The modularity of our model allows different generative functions to be swapped in and out easily. This property of the design has allowed us to use our Python implementation as a tool for measuring the performance of different generative algorithms at the task of interactive improvisation. These participant studies are an area of ongoing work.

7 Conclusion

We have presented a high-level generative model for traditional jazz improvisation along with some examples of its application. The music generated by our implementations to date has been promising, and we plan to continue development of both our Haskell and Python versions of the system. We hope that our implementations can ultimately be used as adaptive, interactive tools for musicians looking for automated accompaniment and for composers looking for sources of inspiration. We are also currently exploring a broader application of our models to styles other than jazz.

One of the interesting areas of future application for our Python implementation is that it allows the evaluation of different generative models through participant studies. In one experimental design that we have explored already with a pilot study, participants interact with a 3-part generative system under several different soloing algorithm settings. After each improvisation session with the system, participants rate the soloing algorithm's performance according to a variety of criteria (creativity, coherency, appropriateness of responses, and so on). We plan to run full studies soon with this system as a way to quantitatively measure the performance of different soloing algorithms.

One shortcoming of this system is that it handles lead sheet segments in a very discrete way that is uniform across all of the parts. Planning may occur at different levels of granularity for different instruments or parts, so some instruments may be better modeled with smaller or larger segments than others. Additionally, the modeling of "listening" to the shared musical output could be more continuous rather than at junctions of lead sheet segments. Using a threaded approach similar to that of our interactive Python implementation, we feel this would be possible while still using a more discrete approach to generating output.

For our Haskell implementation, a monadic implementation that encapsulates behaviors like the passing of random generators might present a cleaner and interface. This would allow us to wrap many common operations involved in generating music, such as the selection of one pitch from a pitch space, finding collections of pitches that instantiate an abstract chord, determine membership of pitches to a key/mode, and so on. However, we also feel this type of design choice is best left until we have a more complete picture of the possible behaviors that would need encapsulating, including what kinds of features constitute a sufficiently generalized state data structure.

Our models are also applicable to styles beyond jazz. Other improvisational music that has a turn-based or segmented structure can also be implemented, although the complexity of the style would potentially be limited by the algorithms' ability to keep up with real-time demands. We have done some preliminary exploration of using the same techniques

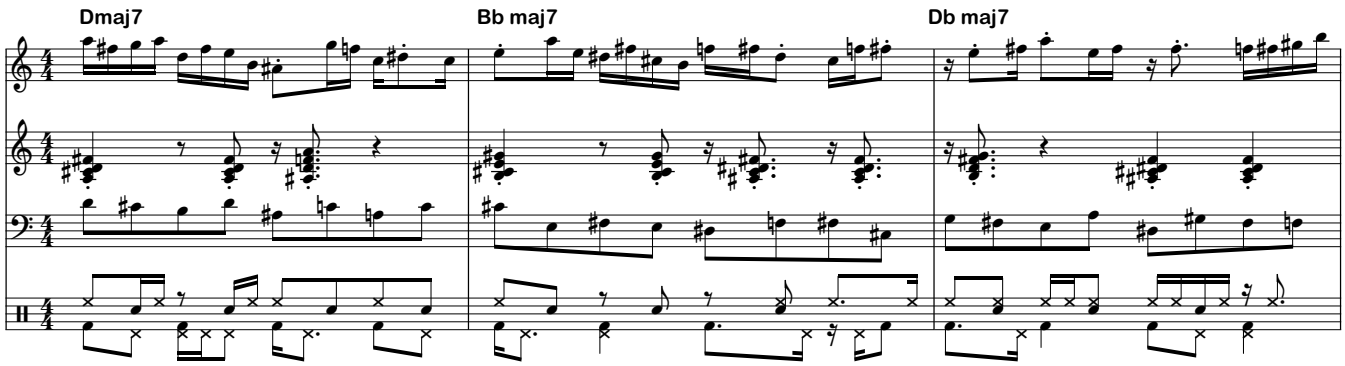

Figure 13. Excerpt from a 4-part system that also includes a generative function for drums. The lead sheet was also algorithmically generated and is technically infinite; only the first three measures are shown here.

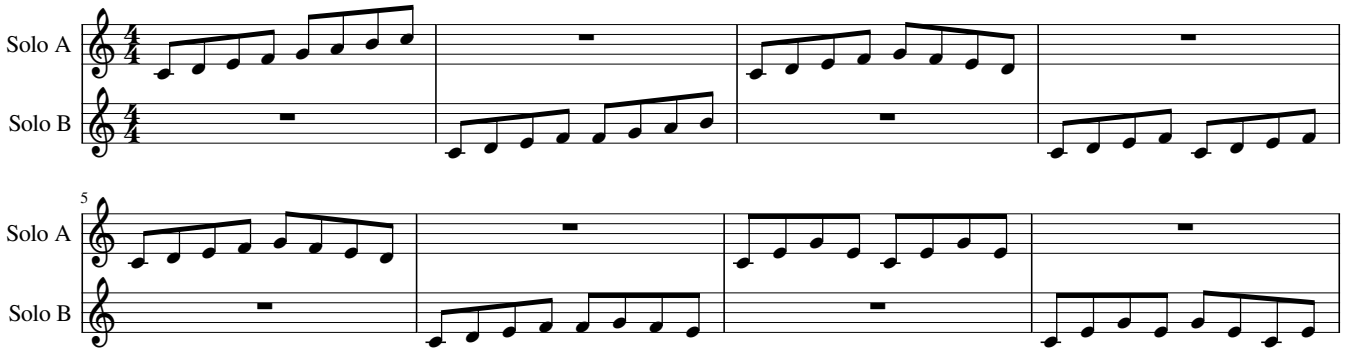

Figure 14. Example of simple solo trading generated by our model. In this very simplified case, soloist A simply chooses one of three possible phrases to play, and soloist B recombines what it "hears" by re-arranging chunks of 4 notes to fill its segment.

to generate ambient background music (roughly in the style of some old game soundtracks) and call-and-response systems similar to Gregorian chant. We are also exploring the use of real-time key detection as a way to govern the changes in harmonic context rather than an existing lead sheet.

Acknowledgments

This work was funded in part by DARPA Grant W911NF-16-1-0567, which is part of the Communicating with Computers program.

References

[1] Google AI. 2019. Magenta. https://magenta.tensorflow.org/

[2] Paul F. Berliner. 1994. *Thinking in jazz : the infinite art of improvisation.* University of Chicago Press.

[3] John A. Biles. 1994. GenJam: A Genetic Algorithm for Generating Jazz Solos. In *Proceedings of the International Computer Music Conference.* 131–137.

[4] Parag Chordia, Avinash Sastry, and Sertan Senturk. 2011. Predictive Tabla Modeling using Variable-length Markov and Hidden Markov Models. *Journal of New Music Research* 40, 2 (2011), 105–118.

[5] Douglas Eck and Jürgen Schmidhuber. 2002. Learning the long-term structure of the blues. In *In Proceedings of the International Conference on Artificial Neural Networks.* 284–289.

[6] Judy A. Franklin. 2004. Recurrent Neural Networks and Pitch Representations for Music Tasks. In *Forida AI Research Symposium.*

[7] Jon Gillick, Kevin Tang, and Robert M. Keller. 2009. Learning Jazz Grammars. In *Proceedings of the Sound and Music Computing Conference.* 125–130.

[8] Paul Hudak et al. 2019. Euterpea. http://hackage.haskell.org/package/Euterpea

[9] PG Music Inc. 2019. Band in a Box. https://www.pgmusic.com/

[10] P.N. Johnson-Laird. 2002. How Jazz Musicians Improvise. *Music Perception* 19, 3 (2002), 415–442.

[11] Robert M. Keller. 2018. Impro-Vizor. https://www.cs.hmc.edu/~keller/jazz/improvisor/

[12] Fred Lerdahl and Ray S. Jackendoff. 1996. *A Generative Theory of Tonal Music.* The MIT Press.

[13] Martin Norgaard. 2011. Descriptions of Improvisational Thinking by Artist-Level Jazz Musicians. *Journal of research in Music Education* 59, 2 (2011), 109–127.

[14] Martin Norgaard. 2013. Testing cognitive Theories by Creating a Pattern-Based Probabilistic Algorithm for Melody and Rhythm in Jazz Improvisation. *Psychology: Music, Mind, and Brain* 23, 4 (2013), 243.

[15] Tomasz Oliwa and Markus Wagner. 2008. Composing Music with Neural Networks and Probabilistic Finite-State Machines. In *Proceedings of the Conference on Appications of Evolutionary Computing.* 503–508.

[16] François Pachet. 2002. The Continuator: Musical Interaction With Style. In *Proceedings of the International Computer Music Conference.* 2011–218.

[17] Jeff Pressing. 2001. *Improvisation: Methods and Models.* Oxford Scholarship Online.
[18] Donya Quick. 1997. Improvising Jazz with Markov Chains.
[19] Donya Quick. 2015. Composing with Kulitta. In *Proceedings of International Computer Music Conference.* 306–309.
[20] Donya Quick. 2016. Learning Production Probabilities for Musical Grammars. *Journal of New Music Research* 45 (2016), 295–313. Issue 4.
[21] Martin Rohrmeier. 2011. Towards a Generative Syntax of Tonal Harmony. *Journal of Mathematics and Music* 5, 1 (2011), 35–53.
[22] Gerard Roma and Perfecto Herrera. 2010. Graph grammar representation for collaborative sample-based music creation. In *5th Audio Mostly Conference.* ACM, Article 17, 8 pages.
[23] Dana Ron, Yoram Singer, and Saftali Tishby. 1996. The Power of Amnesia: learning Probabilistic Automata with Variable Memory Length. *Machine Learning* 25 (1996), 117–149.
[24] John Sloboda. 1988. *Generative Processes in Music: The Psychology of Performance, Improvisation, and Composition.* University of Oxford Press.
[25] Dimitri Tymoczko. 2006. The Geometry of Musical Chords. *Science Magazine* 313, 5783 (2006), 72–74.
[26] Christopher W. White. 2013. Some Statistical Properties of Tonality, 1650-1900.